Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 9 (2012) 1363 – 1370

Procedia Computer Science

International Conference on Computational Science, ICCS 2012

Reachability Analysis of Concurrent Boolean Programs with Symbolic Counter Abstraction

Junyan Qian, Min Li, Lingzhong Zhao*

Department of Computer Science and Engineering,GuiLin University of Electronic Technology,GuiLin541004,China

Abstract

A basic problem in software model checking is the choice of a model for software. Boolean program is the most popular representation and is amenable to model checking. We show how to apply counter abstraction to real world concurrent Boolean programs to eliminate state redundancy. We present a model checking algorithm for thread-state reachability analysis of concurrent Boolean programs, constructing Karp-Miller coverability tree directly on a Boolean program. And finally evaluate the performance of the approach using a substantial set of Boolean program benchmarks.

Keywords:Concurrent Boolean Programs; Symbolic Counter Abstraction; Model Checking; Thread-state Reachability; Karp-Miller Coverability Tree

1. Introduction

Model checking [1] is an automated verification technique to prove the correctness of the finite-state concurrent systems, by searching the state space of a system to determine if the system property holds. Model checking has been successfully used for verification of concurrent systems, such as integrated circuits, communication protocols. However, the main obstacle of model checking concurrent software is the state explosion problem: the number of reachable states of a program grows exponentially with the number of concurrent threads. As a result, straightforward implement of model checking on concurrent software can need excessive memory and computation resources, which leads to the efficiency of verification is very low. To address the problem, a series of methods have been proposed, including symbolic model checking[2], abstraction[3], partial order reduction[4], symmetry reduction[5]and so on, one of the most effective way among these techniques to alleviate the state space explosion is abstraction, especially predicate abstraction[6]is the most successful promoted by Microsoft Research. Boolean programs [7] are obtained from source programs using predicate abstraction, i.e., all variables are in type of Boolean. The verification of Boolean program is the bottleneck in predicate abstraction refinement framework.

Multi-threaded concurrent software often exhibits symmetry, finding a way to reduce the verification complexity according to the nature: counter abstraction. The traditional global state is represented by a vector of local states,

*Corresponding author. Tel.: +86_773_229_1629; fax: +86_773_560_1330.
E-mail address: Zhaolingzhong163@163.com

doi:10.1016/j.procs.2012.04.150

now introducing a counter for each local state, each counter records the number of threads in the current local state, and a vector of counters represents global state of a system. Counter abstraction turns the state space size of concurrent systems grows exponentially in n into polynomial in n. Since a local state is defined as a valuation of all thread local variables of a thread, the number of local states grows exponentially with the number of thread-local variables. As a result, the number of local states generated is very large, probably resulting in many millions of local states. Introducing a counter for each local state is impractical but only for tiny program.

To solve the problem, we present a solution for the thread-state reachability analysis of symmetric concurrent Boolean program. First, instead of statically translating each statement s of the input program into counter changes, do the translation on the fly. This way requires executing s only in the narrow context of a given and reachable local state. In addition, in a global state only keep counters for those local states that at least one thread exists in. Because $l \gg n$ (l, n represent respectively the number of local states and that of threads), in every system state the value of most counters are zero, removing the zero-valued counters will save huge storage space. Second, construct Karp-Miller coverability tree directly on a Boolean program. Since the problem of thread-state reachability can reduce to the coverability problem of VASS (vector addition systems with states) [8], and the coverability problem of VASS can be determined by building a Karp-Miller [9] coverability tree for it. The advantage of constructing Karp-Miller tree directly on Boolean program is to avoid the blowup that Boolean program translation into VASS entails. Finally, evaluate the performance of the method on a set of Boolean program benchmarks.

2. Preliminaries

2.1. Concurrent Boolean Program

Boolean programs are obtained by applying predicate abstraction to the original program P, all variables are Boolean variables. In order to get sound verification of the reachability problem, constructing Boolean program by over-approximate the behavior of P, which may appear spurious paths, this can refine the abstraction further by adding predicates, then detect and eliminate spurious paths, this process is counterexample-guided abstraction refinement(CEGAR)[10].

The Boolean program syntax has been detailed in [7]. SATABS extends the semantics for sequential Boolean programs with the following four instructions to support concurrency:

- The instruction start_thread goto l creates a new thread that starts execution at the program location l. It gets a copy of the local variables of the current thread, which continues execution at the proceeding statement.
- The instruction end_thread terminates the current thread, i.e., has no successor state.
- The instruction atomic_begin prevents the scheduler from a context switch to other thread.
- The instruction atomic_end allows any thread to be executed.

Let P, PC, L, V_s, V_l, V be a Boolean program, program counter, program location, the set of shared variables, the set of thread-local variables, the set of program variables respectively, where $V = V_s \cup V_l$.

Definition 1. An explicit state τ of a Boolean program is a triple $<n, PC, \Omega>$, where $n \in \mathbb{N}$ is the number of threads, PC: $\{1,...,n\} \mapsto L$ is the vector of program locations, Ω: $V_s \cup (\{1,\ldots,n\} \times V_l) \mapsto B \cup \{\star\}$ is the valuation of the program variables.

Definition 2. $<PC, \Omega>$ with $PC \in L$, $\Omega \mapsto B \cup \{\star\}$ is called a thread state. It is a valuation of the program counter, the local variables of a particular thread, and the globally shared variables.

We use $\eta \to \xi$ to denote a transition from thread state η to thread state ξ. The conditions on the transition $\eta \to \xi$ for each instruction are shown as follows:

$$skip: \quad \frac{P(PC) = \text{skip}}{< PC, \Omega > \to < PC + 1, \Omega >} \tag{1}$$

$$goto\ l_1,\ldots,l_k: \quad \frac{P(PC) = \text{goto } l_1,\ldots,l_k}{< PC, \Omega > \to < l_i, \Omega >} \qquad i \in \{1,\ldots,k\} \tag{2}$$

$$x_1,\ldots,x_k := e_1,\ldots,e_k\ constrain\ e: \quad \frac{P(PC) = x_1,\ldots,x_k := e_1,\ldots,e_k \text{constrain } e, \Omega' = [x_1 / [\![e_k, \Omega, \iota]\!]] \ldots [x_k / [\![e_k, \Omega, \iota]\!]], [\![e, \Omega, \Omega', \iota]\!]}{< PC, \Omega > \to < PC + 1, \Omega' >} \tag{3}$$

$$start\_thread: \quad \frac{P(PC) = \text{start thread } l}{< PC, \Omega > \rightarrow < PC+1, \Omega >} \qquad \frac{P(PC) = \text{start thread } l}{< PC, \Omega > \rightarrow < l, \Omega >} \tag{4}$$

2.2. Counter Abstraction

Counter abstraction [11] can be seen as a form of symmetric reduction that presented by Pnueli et al. in the verification of liveness properties of parameter system, the main idea is to use an abstract state replacing the concrete state of a concurrent system that consists of many similar components, each abstract state is a k-tuple of integer, denoted by $(c_1, c_2,..., c_k)$, c_j counts how many processes are currently in the j-th state. A transition from local state A to local state B can be translated a decrement of the counter for A and an increment of that for B. This transformation can be implemented statically in the text of symmetric program P before creating the Kripke model, the resulting counter-abstracted program $\hat{P}$ generates a Kripke structure $\hat{M}$ whose reachable is isomorphic to that of the traditional quotient structure $\bar{M}$, so that can be model checked without considering further symmetry.

Counter abstraction technique makes a problem of size exponential in n reduces to one of size polynomial in n, improving the efficiency of the verification. For any given Boolean program, l is a constant, seen from a theoretical view, seems to have solved the state space explosion problem.

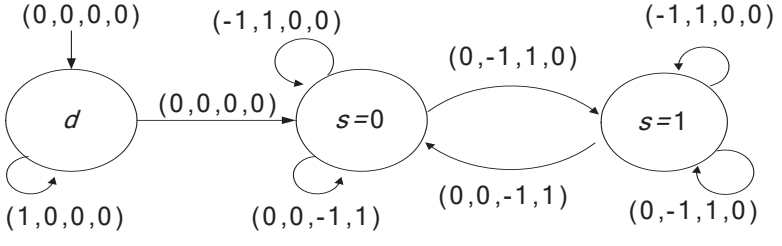
However, for concurrent software, the thread behavior is given in the form of a program that control thread-local variables, a local state is defined as a valuation of all thread-local variables, which is incompatible in practice with the idea of counter abstraction: the number of the local states generated is very large. For example, a Boolean program with five thread-local variables and the PC with range {1..10} can generate 2^5*10=320 local states. In practice, concurrent Boolean programs generally have dozens of thread-local variables and thousands of lines, which can generate millions of local states. As a result of the local state space explosion problem, the state space of the counter-abstracted program is of size $\Omega(n^{2^{|V|}})$, doubly-exponential in the number of thread-local variables. Therefore, the traditional counter abstraction is unfit to general programs but for many programs.

2.3. Karp-Miller Coverability Tree

We use S, $\mathbb{Z}^m$ and $\mathbb{N}^m$ respectively denote the set of states, the set of m-tuple of integers and the set of m-tuple of non-negative integers. An m-dimensional VASS is a finite-state machine whose edges are labeled with $v \in \mathbb{Z}^m$, denoted by $W = (d, \delta)$, where $d \in S \times \mathbb{N}^m$ is the initial configuration and δ: $S \rightarrow S \times \mathbb{Z}^m$ is the transition function. A configuration of a VASS is denoted by (q, x), where q is a state, x is an m-tuple of non-negative integers. There is a transition $(q, x) \rightarrow (q', x')$ if $(q', v) \in \delta(q)$ and $x' = x + v$. A Configuration (q, x) is reachable if there is a sequence of transitions start at d and end at (q, x). The coverability problem of VASS asks whether a given configuration (q, x) is covered by the VASS, i.e., whether a configuration (q, x') is reachable such that $x' \geq x$. Figure 1 shows an example of VASS.


```
decl s:=0
main () begin
0: goto 1
1: s: =1
2: s: =0
3: end
```

(a) Boolean program



(b) VASS

Fig.1. (a) Boolean program; (b) VASS

Theorem 1([12]). The coverability problem for VASS is decidable.

The thread state reachability problem can be reduced to VASS coverability problem, for this reduction, m local states corresponding to an m-dimensional vector, the value of i-th component of VASS configuration vector

corresponding to the number of threads in state i. As a result, the time complexity of thread-state reachability problem is equivalent to that of the coverability problem for VASS that are polynomial.

Corollary 2. The thread-state reachability problem for replicated finite-state programs is decidable.

Karp-Miller tree can compactly represent the set of covered configurations of a VASS. The algorithm as follows:

- The root of the tree is labeled by the initial configuration of VASS.
- For any two nodes ξ and η in the tree, if there is a path from ξ to η, denotes as $\xi \prec \eta$; if there is an edge from ξ to η, then η is a successor of ξ. A node without successors is called an end node.
- Each node η of the tree is labeled with $l(\eta) = (q, x)$, where q is a state, x is m-dimensional vector whose coordinate are elements of $\mathbb{N} \cup \{\omega\}$ (ω is any natural number). For any node ξ labeled with $l(\xi) = (p, y)$ differs from η, we say $l(\eta)$ covers $l(\xi)$ if $p=q$ and for each i-th coordinates of vectors x and y, such that $x_i \geq y_i$.
- For some node η, $l(\eta) = (q, x)$: η is an end node if there exist some node ξ, $\xi \prec \eta$, $l(\xi) = l(\eta)$; otherwise, ξ is a successor of η, $l(\xi) = (q', x')$. Determines each i-th coordinates of vector x' as followings (We call it the ω procedure): (1) if there is a path $\xi \prec \eta$ and $l(\xi) = (s, y)$, such that η covers ξ and $y_i < (x + v)_i$, then $x'_i = \omega$, ω is a symbol such that $z \in \mathbb{Z}$, $\omega > z$ and $\omega + z = \omega$;(2) if there is no such node ξ, then $x'_i = (x + v)_i$. This process eventually reaches a fix point. If there is a label (q, v) such that $x \leq v$, then the configuration (q, x) is covered by the VASS.

Figure 2 depicts the Coverability Tree of the example in Figure 1:

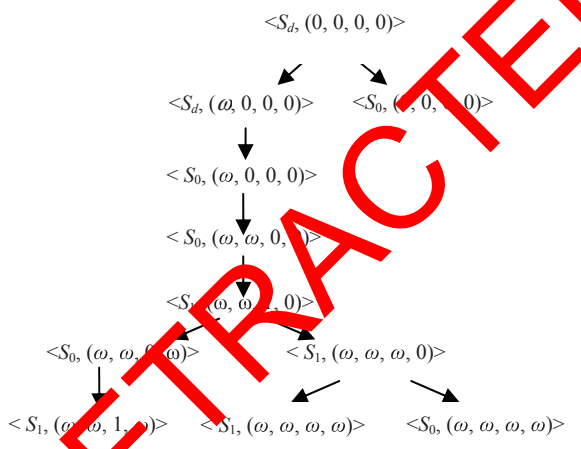


Fig. 2. Corresponding Karp-Miller tree for Figure 1

3. Symbolic Counter Abstraction

To solve the state sapce explosion problem faced during the implementation of the traditional counter abstraction, we give a solution for the thread state reachability analysis of concurrent Boolean program. First, instead of statically translating each statement s of the input program into counter updates, do the translation on the fly. In a global state only keep counters for those local states that at least one thread resides in. Because $l \gg n$, in every system state most counters are zero; Second, construct Karp-Miller coverability tree directly on a Boolean program. The advantage is to avoid the blowup that Boolean program translation into VASS entails. As a result, the worst-case size of the Kripke structure of the counter-abstracted program is reduced from n^l to $n^{min\{n, l\}}$, completely eliminating the sensitivity to the local state space explosion problem. We first describe a data structure used to store system states compactly before describing the symbolic state space exploration algorithm that implements this method.

3.1. Symbolic representation

Boolean program make heavy use of data nondeterminism, the nondeterministic Boolean value is denoted by$\star$. In practice, enumerating all possible values of expression associated with $\star$ is infeasible, a better method is to

interpret ✶ symbolically, this interpretation is not only compatible with Boolean program encodings based on BDDs, but can also be combined well with counter abstraction. Our approach is to count sets of local states, rather than individual local state.

To formalize the state representation, let L be the set of local states, i.e., $|L|=l$. An abstract global state can be expressed as:

$$<S, (L_1, n_1), \ldots, (L_k, n_k)> \quad (5)$$

In this expression, S represents valuations of the shared variables, L_i represents the set of local states, $L_i \subseteq L$, n_i represents a counter, $n_i \geq 1$, $n_i \in \mathbb{N} \cup \{\omega\}$, (L_i, n_i) denotes there are n_i threads in the local state L_i. For instance, the abstract global state $< (\{x=1\}, 2), (\{x=0\}, 4)>$ represents these concrete states where 2 threads satisfy $x=1$, whereas the remaining 4 threads satisfy $x=0$. The semantics of this representation is given by the set of concrete states that expression (5) represents, namely the states of the form $(s, l_1, \ldots, l_n)$such that:

(a) $s \in S$,

(b) $n = \sum_{i=1}^{k} n_i$, (6)

(c) there exists a partition $\{I_1, \ldots, I_k\}$ *of* $\{1, \ldots, n\}$ *such that for all* $i \in \{1, \ldots, n\}$, $|I_i| = n_i$ *and for all* $j \in I_i$, $l_j \in L_i$.

In traditional counter abstraction, $k=l=|L|$ and do not require $n_i \geq 1$, thus suffer from the potential redundancy of n_i being 0 for most i.

In addition, representation (5) has a problem. When communicate data between threads, Boolean programs can introduce such constraints between shared and thread-local variables, such as introduced by an assignment of the form *shared*: =*local*, are inexpressible. The reason is that neither S nor the sets L_i are defined by expressions over both shared and thread-local variables. In order to make this constraint expressible, we must treat certain assignments and related statements specially.

Definition 3. A splice state is a predicate f over shared V_s and thread-local variables V_l such that

$$(\forall V_s. f) \wedge (\exists V_l. f) \neq f$$

Splice states result from communication. Split the current thread state before executing a splice statement as follows:

$$Image(T) = Image(T|v=0) \vee Image(T|v=1)$$

The worst-case cost of state splitting is exponential in the number of splice variables. However, we observed in experiments: (1) there is relatively few splice statements; (2) very few splice variables in a splice statement (one or two); (3) relatively proportion of cofactors encountered during the exploration is actually unsatisfiable and does not produce new states. As a result, the decomposition never suffers from the potential combinatorial explosion.

3.2. Symbolic State Space Exploration

We present a symbolic model checking algorithm for thread-state reachability analysis of concurrent Boolean programs, constructing Karp-Miller coverability tree directly on a Boolean program (Fig 3). The algorithm uses the state representation described section 3.1, mainly computes the counter-abstracted set of states reachable from a given set of initial state, and checks whether the new node covers previously discovered node. To make the search efficient, keep a separate copy of those discovered nodes that are maximal as to the covering partial order. New labels are compared against these maximal nodes only.

More precisely, when a new node η is found, the algorithm first checks whether some previously discovered node covers η. if so, then discard η. To check this, only compare η with the maximal elements of each chain in the covering partial order. Conversely, the algorithm checks whether η covers some previously discovered node θ. if so, then removed θ from the unexplored list. The algorithm as follows:

```
1: Initialize: R :={< S0, (L0, n0) >}; insert<S0, (L0, n0)>into WorkList        // n0 threads in L0
2: While WorkList ≠ Ø do
3:   pop τ =<S, F> (F= {(L1, n1),…,(Lk, nk)})from WorkList
4:   for i ∈ {1,...,k}do
5:     T :=< S, Li>                                                  // extract i-th thread state from τ
6:     for v∈all valuations of Splice Variables (T) do
7:       T'=<S', L' >:= Image (T | v)                                // compute successor thread state
8:       if L'≠ Li then
9:        τ':= UPDATECOUNTERS (S, S', F, i, L')>                     // build new system state τ'
10:       τ'':= UPDATECOVER (τ')                                     // build covered system state τ"
11:       if τ''∉ R then
12:         R: = R∪τ'                                                // if new, store τ' as reachable
13:         insert τ' into WorkList
```

Fig.3. Reachability Analysis

The algorithm (in Fig 3) expands unexplored system states from a worklist. The loop in line 4 iterates over all pairs (L_i, n_i) contained in the popped state τ. The next and crucial step is to compute the successor thread states (lines 6-7). After computing the image for each cofactor, the algorithm constructs the respective system state for it (lines 8-9).

The UPDATECOUNTERS function (see Fig 4(a)) determines the new set of F' according to the local state part L' of the newly computed thread state. If $n_i=\omega$, then n_i is never updated since $\omega - 1=\omega$. If $n_i\neq\omega$ and no more threads reside in the state L_i, then eliminate the i-th pair (line 4). If the new local state L' was already present in the system state, if $n_j\neq\omega$, $n_i=\omega$ and $S'=S$, then n_j doesn't change, since $\omega + 1=\omega$ (line 8); if $n_i\neq\omega$, then n_j is incremented (lines 9-11). Otherwise the state is inserted with counter value 1 (line 12).

The procedure UPDATECOVER (see Fig 4(b)) introduces ω for the thread states that can be reached. The idea is to look for cycles in the execution path that spawn new threads infinitely. That is, we first walk backwards towards the initial state (line 4) and check if any preceding state is subsumed by the state τ. If such a state τ' has been found (line 5), any pair (L_i, n_i) that occurs in the new state τ with a higher counter value than in the preceding state τ' is replaced by the pair (L_i, ω) (lines 6-8).

Finally, the algorithm adds the states encountered for the first time to the set of reachable states, and to the worklist of states to expand (lines 12-13 in Fig 3).


```
1: procedure UPDATECOUNTERS (S, S', F, i, L')
2: let (Li, ni) be the i-th pair in
3: if ni≠ω then
4:F':= F'\{(Li ,ni)} ∪(ni >1 ? {(Li, ni −1)}: Ø)
5: if ∃j. (L', nj) ∈F then
6: if nj≠ω then
7:  if ni=ω且 S'=S then
8: F':= F'\ {(L', nj)} ∪ {(L', ω)} else
9: F':= F'\ {(L', nj)} ∪ {(L', nj + 1)}
10: else if ni=ω且 S'=S then
11: F':= F'\ {(L', ω)}
12:  else
13: F':= F'\ {(L', 1)}
14: return< S', F'>
```

(a) Build Coverability Tree

```
1: procedure UPDATECOVER (τ)
2:  σ=τ
3: let (Li, ni) be the i-th pair in σ=<S, (L1, n1),…,(Lk, nk)>
4:  for all τ' ∈ Pred(τ) do
5: if τ ⊆ τ and S'=S then
6: let (L'j,n'j) be the i-th pair inτ'=<S',(L'1, n'1),…(L'k, n'k)>
7:  I: = {i | L'j=Li, n'j< ni}
8:     σ:=σ[(Li, ni)/ (Li, ω), i∈I]
9: return σ
```

(b) Build Covering State

Fig.4. (a) Build Coverability Tree; (b) Build Covering State

4. Experimental Evaluation

We have implemented the algorithm presented in this paper in a tool of reachability analysis of concurrent Boolean program called BOOM, the concurrent benchmarks used is drawn from a large and diverse set of benchmark Boolean programs generated by SATABS[13]that abstract Linux and Windows kernel components using SLAM tool[14]. The experimental setup is as follows: we run full reachability analysis with n_0=1 initial thread, and then increase the bound N until the tool times out. The timeout is set to 720s and the memory limit to 12GB. The experiments are performed on Linux operating system.

The main goal of this paper is the symbolic analysis of Boolean programs. In the same benchmarks, we compare the running time of the symbolic counter abstraction with that of a plain symbolic implementation in BOOM that ignores the symmetry, shown in Fig 5(a). Counter abstraction gets the improvement in scalability, when the number of threads is small, traditional Model Checking is faster, but the complexity of counter abstraction is polynomial; in fact, our algorithm can verify many instances for 7 or more threads. Overall, the running time of symbolic counter abstraction is faster. For the thread-state reachability analysis with unbounded thread creation, the experiment shows the running times using Karp-Miller method are very small (Fig 5(b)).

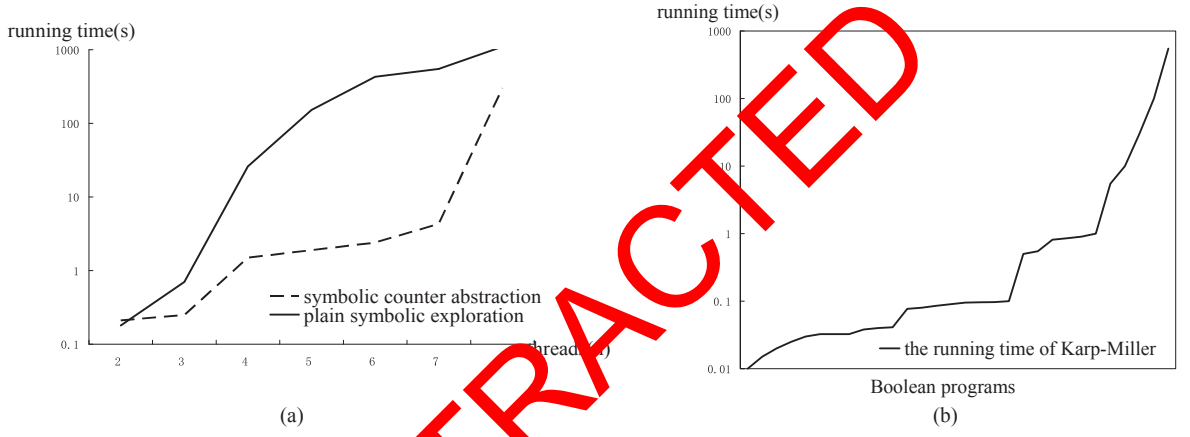


Fig.5. (a) Running time of symbolic counter abstraction vs. plain exploration; (b) Running time of Karp-Miller method for various Boolean programs

5. Conclusion

We present a model checking algorithm based on symbolic counter abstraction for thread-state reachability analysis of concurrent Boolean programs, construct Karp-Miller coverability tree directly on a Boolean program, which avoids effectively the local state space explosion problem facing that Boolean program translation into VASS, experimental results show the effectiveness of the method.

In concurrent software verification, counter abstraction has an improvement in scalability. Since the state space size of the counter-abstracted program is polynomial, which reduces greatly the complexity of the verification. Symbolic counter abstraction promoted the verification of symmetric concurrent systems that described using Boolean programs. In future work we will consider how to combine it with other techniques to achieve more effective compression, such as partial order, and research how to apply it to reachability analysis of concurrent systems of parameterized systems or unbounded dynamic thread creation.

Acknowledgements

This work is supported by Natural Science Foundation of China (No. 61063002, No. 60803033), Guangxi Natural Science Foundation of China (No.2011GXNSFA018164, No.2011GXNSFA018166), China Postdoctoral Science Foundation (No.20090450211), Guangxi Graduate Innovation Foundation (No. 2011105950812M19). We

sincerely thank all the colleagues in Software Technology Institute for helping and inspiring discussions on reachability analysis of concurrent Boolean programs.

References

1. E. Clarke, O. Grumberg, D. Peled.Model Checking.Cambridge: MIT Press,1999.
2. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, etc.NuSMV 2:An OpenSource Tool for symbolic model checking.In Proceedings of the 14th International Conference on Computer Aided Verification. LNCS 2404, Copenhagen:Springer-Verlag,2002,359−364.
3. P. Cousot.The Role of Abstract Interpretation in Formal Methods.In Proceedings of the 5th IEEE International Conference on Software Engineering and Form- al Methods. London: IEEE,2007,135−140.
4. V. Kahlon, C. Wang, A. Gupta. Monotonic Partial Order Reduction: An Optimal Symbolic Partial Order Reduction Technique. In Proceedings of the 21st Inte- rnational Conference on Computer Aided Verification. LNCS 5643, Grenoble:Springer-Verlag,2009,398−413.
5. E. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting Symmetry in Temporal Logic Model Checking. Formal Methods in System Design,1996,9(1-2):77-104.
6. K. Lahiri, R. Bryant and B. Cook.A Symbolic Approach to Predicate Abstraction.In Proceedings of the 15th International Conference on Computer Aided Verification. LNCS 2725, Boulder:Springer-Verlag,2003,141-153.
7. T. Ball, S. Rajamani. Bebop: A Symbolic Model Checker for Boolean Programs. In Spin Model Checking and Software Verification, LNCS1885, Springer-Verlag,2000,113-130.
8. T. Brázdil, P. Jančar and A. Kučera. Reachability Games on Extended Vector Addition Systems with States. In Proceedings of the 37th International Colloquium on Automata, Languages and Programming. LNCS 6199, Bordeaux:Springer-Verlag,2010,478-489.
9. K. N. Verma and J. Goubault-Larrecq.Karp-Miller trees for a branching extension of VASS. In Discrete Mathematics and Theoretical Computer Science, 2005,7(1):217-230.
10. R. Kurshan. Computer-Aided Verification of Coordinating Processes.Princeton University Press, 1995.
11. A. Pnueli, J. Xu, and L. Zuck. Liveness with (0,1,∞)-counter abstraction. In Proceedings of the14th International Conference on Computer Aided Verification.LNCS 2404, Copenhagen:Springer-Verlag,2002,107-122.
12. R. Karp and R. Miller. Parallel program schemata. In Computer and System Sciences,1969,3(2):147-195.
13. E. Clarke, D. Kroening, N. Sharygina and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In Tools and Algorithms for the Construction and Analysis of Systems, LNCS3440, Springer-Verlag,2005,570-574.
14. T. Ball and S. Rajamani. The SLAM project: debugging system software via static analysis.In Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, 2002,37(1):1-3.